UNIVERSIDAD REY JUAN CARLOS

TRABAJO FIN DE GRADO

Aplicación para la compartición segura de ficheros

Autor:
Sergio Merino Hernández

Tutor:
Dr. Gorka Guardiola Múzquiz

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
GRADO EN INGENIERÍA EN TELEMÁTICA

28 de febrero de 2018

Resumen

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Agradecimientos

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Índice general

Índice de figuras

Índice de cuadros

Lista de Abreviaciones

AES	**A**dvanced **E**ncryption **S**tandard
API	**A**pplication **P**rogramming **I**nterface
ART	**A**ndroid **R**un**t**ime
BC	**B**ouncy **C**astle
CBC	**C**ipher **B**lock **C**haining
CFB	**C**ipher **F**eed**b**ack
ECB	**E**lectronic **C**ode**b**ook
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
IV	**I**nitialization **V**ector
JCA	**J**ava **C**ryptography **A**rchitecture
JCE	**J**ava **C**ryptography **E**xtension
JIT	**J**ust **I**n **T**ime
JVM	**J**ava **V**irtual **M**achine
MAC	**M**essage **A**uthentication **C**ode
MCD	**M**áximo **C**omún **D**ivisor
mcm	**m**ínimo **c**omún **m**últiplo
OFB	**O**utput **F**eed**b**ack
PSS	**P**robabilistic **S**ignature **S**cheme
RSA	**R**ivest - **S**hamir - **A**dleman
SSA	**S**ignature **S**cheme with **A**ppendix

For/Dedicated to/To my...

Capítulo 1

Introducción

1.1. Motivación

Las comunicaciones seguras nacen del deseo de protegernos: de proteger con quién nos comunicamos y el qué comunicamos.

De este deseo surgen multitud de protocolos de seguridad que hoy en día usamos sin darnos cuenta. Desde una simple consulta web hasta la felicitación de Año Nuevo, nuestras comunicaciones pasan por diversas operaciones para preservar su seguridad.

Esta seguridad viene generalmente proporcionada por la confianza que depositamos en ciertas organizaciones, entidades que crean una red de confianza sobre la que se sustenta todo este sistema. Pero, ¿qué sucede si de quién nos queremos proteger es de ellos? ¿Por qué tengo que confiar en que una entidad gubernamental sea la que mantenga la seguridad de mis comunicaciones? ¿Por qué no puedo tener mi propia red de confianza?

Con esta idea comienza el desarrollo de una herramienta que nos permita crear nuestra propia red de confianza, con la que poder mantener comunicaciones seguras.

1.2. Estructura de la memoria

La estructura que se va a seguir en este proyecto es la siguiente:

- En el Capítulo 2 se presentan los objetivos que se persiguen con este proyecto y un modelo de amenaza para el mismo.
- En el Capítulo 3 se explican varios conceptos y tecnologías que ya existen y que se han usado para llevar a cabo el proyecto.
- En el Capítulo 4 se detallan los diferentes prototipos y la arquitectura de seguridad de la aplicación.
- En el Capítulo 5 se exponen los resultados obtenidos con ejemplos y los problemas encontrados durante el desarrollo del proyecto.
- En el Capítulo 6 se finaliza la memoria haciendo una reflexión sobre los resultados y las posibles líneas de desarrollo futuras.

Capítulo 2

Objetivos

2.1. Objetivo general

El objetivo principal es el desarrollo de una aplicación para el sistema operativo Android, que permita a cualquier usuario realizar una comunicación de datos de manera que se preserve la **confidencialidad**, la **integridad** y la **autenticación** de la información transmitida.

2.2. Objetivos específicos

Para abordar el objetivo principal del proyecto, éste se ha dividido en unos objetivos más específicos:

- Proporcionar un canal por el que transmitir la información.
- Desarrollar un esquema que permita cifrar y descifrar la información que queremos transmitir.
- Diseñar una forma de comprobar la integridad y la autenticación de la información.

2.3. Modelo de amenaza

En criptografía, un modelo de amenaza (*threat model*) especifica que tipo de amenazas potenciales a un sistema puede realizar un individuo (o grupo) con unos determinados recursos. En base a estas amenazas, se desarrolla un determinado esquema de seguridad para poder contrarrestarlas.

Nuestro modelo de amenaza será un atacante con conocimientos de criptoanálisis, utilizando para el ataque una máquina con un procesamiento computacional estándar.

Los puntos más vulnerables de nuestro modelo son el hardware y el software, por lo que dejamos fuera a aquellas compañías que han trabajado en su desarrollo, como Intel, Google, Qualcomm, etc.

Debido a los recursos que poseen también dejamos fuera de nuestro modelo a cualquiera de los gobiernos de las grandes potencias del mundo, como China, Corea, Rusia o Estados Unidos.

Un atacante de nuestro modelo de amenaza podría realizar los siguientes tipos de ataque:

- Ataques sobre el texto cifrado – En este tipo de ataques el atacante dispone únicamente del texto cifrado y no tiene acceso al texto plano. Ejemplos de este modelo son los ataques por fuerza bruta, en los que se prueba cada una de las posibles combinaciones para una clave hasta dar con la correcta.
- Ataque de texto plano conocido – En este tipo de ataques se presupone que el atacante tiene acceso a un número limitado de textos planos y sus correspondientes textos cifrados. El objetivo de este tipo de ataques es el de obtener la clave de cifrado a partir de estos pares, con el fin de poder desencriptar futuras comunicaciones.
- Ataque de texto plano selectivo – En este tipo de ataques el atacante puede seleccionar un número indeterminado de textos planos y obtener sus equivalentes cifrados. Con esto, un atacante puede probar distintas combinaciones y encontrar patrones sobre el texto plano para explotar alguna vulnerabilidad.
- Ataque de texto cifrado selectivo – Parecido al anterior, el atacante puede obtener un número indeterminado de textos planos a partir de sus equivalentes cifrados.
- Ataques sobre la clave – En este tipo de ataques el atacante dispone de alguna información acerca de la clave de cifrado.
- Man in the middle
- Ataques de side channel
- Ataques de reply

[25]

Capítulo 3

Estado del Arte

3.1. Java

Java es un lenguaje de programación de propósito general, orientado a objetos y concurrente. Originalmente fue desarrollado por James Gosling, Bill Joy y Guy Steele para Sun Microsystems en 1996 y fue adquirido por Oracle en 2010.

Fue diseñado para que los desarrolladores escribiesen una única vez su programa y pudieran ejecutarlo en cualquier máquina sin necesitar recompilarlo. Esto es posible debido a que las aplicaciones Java son compiladas a *bytecode* que luego es ejecutado en una **Java Virtual Machine (JVM)**, sin importar la arquitectura de la máquina. *[8]*

FIGURA 3.1: Logo de Java *[20]*

3.2. Android

Android es un sistema operativo desarrollado por Google, basado en el *kernel* de Linux. Está diseñado principalmente para dispositivos táctiles, como *smartphones* y *tablets*.

Las aplicaciones de **Android** están escritas en Java. Hasta la versión 4.4 de Android, se utilizaba Dalvik como máquina virtual con la compilación en tiempo de ejecución (JIT) para ejecutar *bytecode* Dalvik, que es una traducción del *Java bytecode*.

Android 4.4 introdujo el ART (Android Runtime) como un nuevo entorno de ejecución, que compila el *Java bytecode* durante la instalación de una aplicación. Desde Android 5.0 se convirtió en la única opción en tiempo de ejecución. *[22]*

FIGURA 3.2: Logo de Android *[19]*

3.3. Bouncy Castle

Bouncy Castle (BC) es una API utilizada en criptografía. Esta API, entre otras cosas, proporciona los siguientes servicios:

- Una API criptográfica *ligera* para Java y C#.
- Un proveedor para Java Cryptography Extension (JCE)[1] y para Java Cryptography Architecture (JCA).

BC está mantenidos por una organización caritativa australiana, conocida como **The Legion of the Bouncy Castle**. *[3]*

3.4. Criptografía de clave simétrica

La *confidencialidad* en las comunicaciones es el objetivo principal que se persigue con los algoritmos de encriptación. La **criptografía de clave simétrica** nos permite comunicarnos con otra persona (o máquina) de manera que un tercero, aun teniendo el mensaje cifrado, no pudiera extraer nada de información de él.

Los algoritmos basados en este modelo utilizan un *secreto* compartido entre los dos extremos que se quieren comunicar. Una vez acordado, nadie que no posea este secreto podría descifrar ningún mensaje enviado por uno u otro extremo. (Figura 3.3)

El problema que tiene este modelo es el intercambio de las claves. Es necesario un canal seguro por el que comunicar las claves y este tipo de algoritmos no proveen ese servicio. *[5]*

Es por ello que normalmente se mezcla este tipo de criptografía con otro conocido como **criptografía de clave pública**, que veremos más adelante.

[1] JCE implementa encriptación, generación y protocolos de establecimiento de claves y algoritmos MAC.

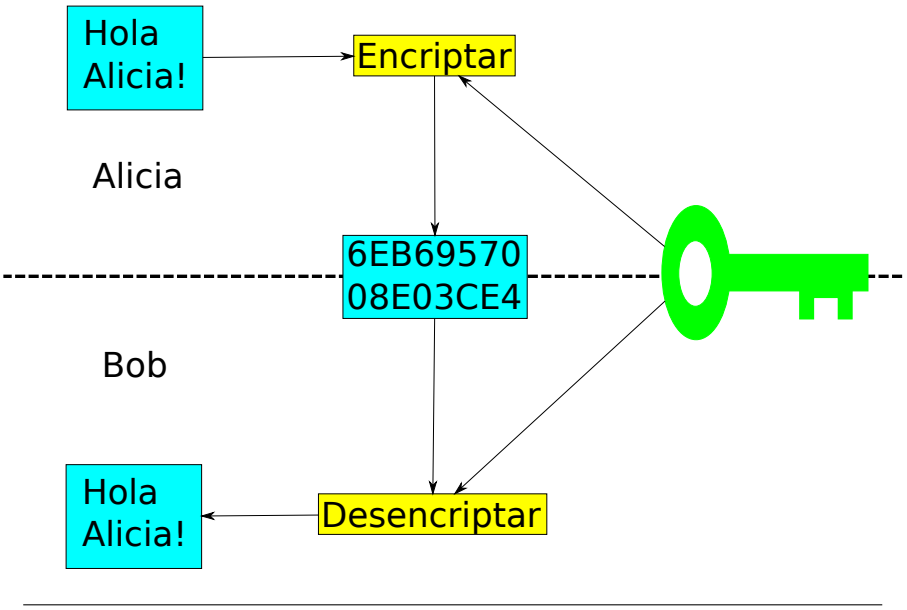

FIGURA 3.3: Esquema general de la criptografía de clave simétrica

3.5. Cifrado por bloques

Dentro de la criptografía simétrica nos encontramos dos grandes algoritmos: los de **flujo** y los de **bloques**. Nos centraremos en el segundo.

Un algoritmo basado en **cifrado por bloques** es aquel que, como su propio nombre indica, realiza un cifrado primero dividiendo el *texto plano* en bloques de un tamaño determinado[2] y luego cifrando por separado estos bloques, dando como resultado un *texto cifrado*.

La mayoria de los **cifradores por bloques** son iterativos, es decir, realizan la misma operación un determinado número de veces o rondas (*rounds*). Esta operación suele ser idéntica en todas las rondas, a excepción de la primera o la última, donde suele ser distinta.

Existen varios modos de operación para llevar a cabo un **cifrado por bloques**: ECB, CBC, OFB, CFB, etc. Cada uno de ellos opera los bloque de diferente forma: algunos utilizan el bloque cifrado anterior para generar el nuevo, otros utilizan combinaciones con estructuras del mismo tamaño que el bloque, etc. *[4]*

3.6. Relleno (Padding)

Relleno (Padding) es el nombre que recibe la técnica que permite en criptografía por bloques expandir el último bloque del mensaje hasta lograr un tamaño deseado.

[2]Cuando el tamaño del fichero que queremos encriptar no es múltiplo del tamaño de bloque, el bloque final tendrá un tamaño diferente al resto. Esto se soluciona con técnicas de padding, lo cual se explica en el siguiente punto.

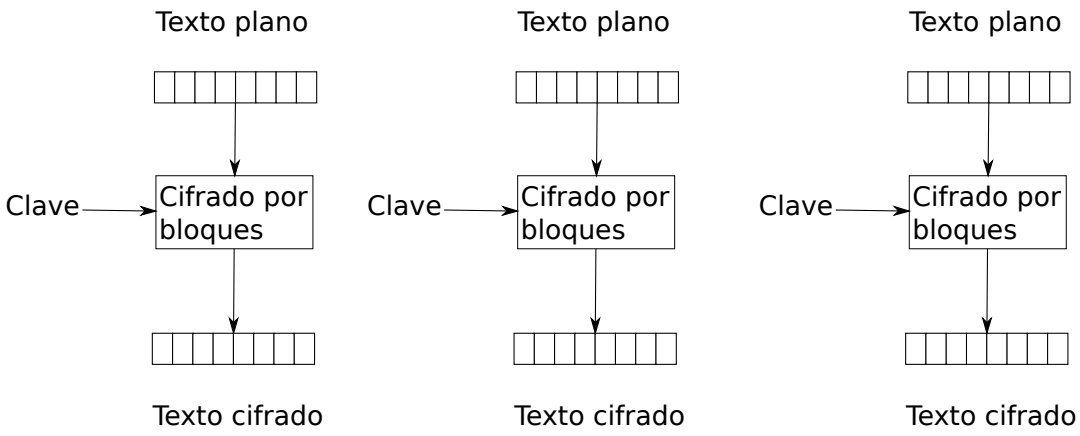

FIGURA 3.4: Cifrado usando el modo Electronic Codebook (ECB)

Es muy común que, en la criptografía por bloques, los fragmentos que encriptamos no tengan la longitud que queremos para nuestro sistema. Para solucionar esto existen multitud de técnicas de **padding**, desde agregar al final del bloque un byte con un cierto valor, hasta simplemente rellenar con ceros.

El requisito indispensable que debe cumplir cualquier técnica de padding es que debe permitir al destinatario diferenciar los bytes del mensaje original de los byte de relleno. *[1]*

3.7. CBC

Cipher Block Chaining (CBC) es uno de los modos de operación para cifrado por bloques que hemos mencionado antes, y el que se ha decidido elegir para este proyecto.

En algunos modos como ECB, dada una clave determinada, cualquier *plaintext* siempre dará como resultado el mismo *ciphertext*. Si, como es nuestro caso, esta característica supone un problema, se opta por otros modos de operación como CBC, el cual soluciona esto. *[23]*

3.7.1. Modo de operación

CBC funciona combinando cada bloque de *plaintext* con el bloque de *ciphertext* justamente anterior. Obviamente, para el primer bloque no se dispone de un bloque cifrado anterior, por lo que se recurre al uso de un **vector de inicialización (IV)**. [3]

Como vemos en la Figura 3.5, en el cifrado con **CBC**, el primer bloque de texto y el IV son sometidos a una operación XOR. El resultado de esta operación se pasa por una función de cifrado[4], la cual nos dará el primer bloque cifrado. Este primer bloque es entonces pasado por un XOR junto con el segundo bloque de texto, y así sucesivamente.

[3]Un IV es un conjunto de bytes aleatorios que se usan para suplir la necesidad de un bloque cifrado anterior al primer bloque. Una muy mala práctica, por no decir prohibida, es la reutilización de un IV, ya que éste debe ser de uso único.

[4]La función de cifrado es la misma para todos los bloques.

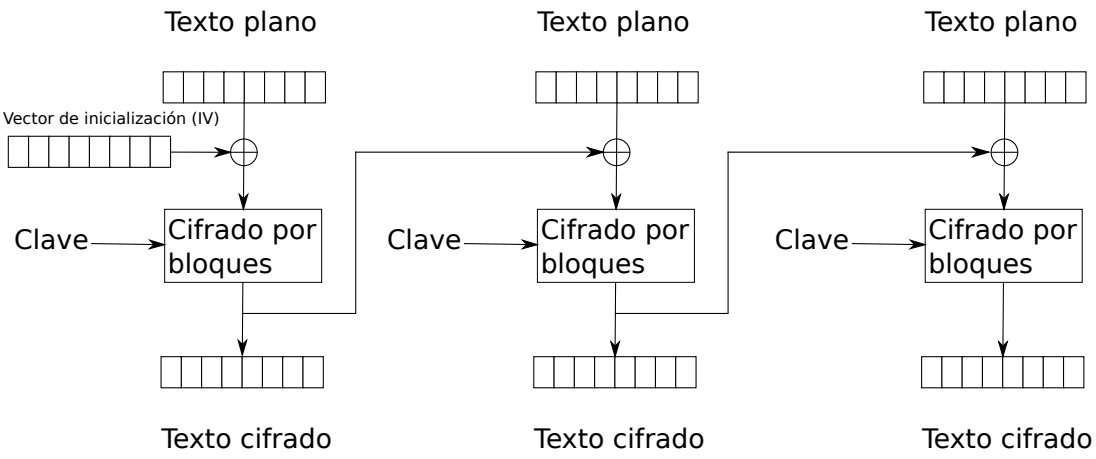

FIGURA 3.5: Cifrado usando el modo Cipher Block Chaining (CBC)

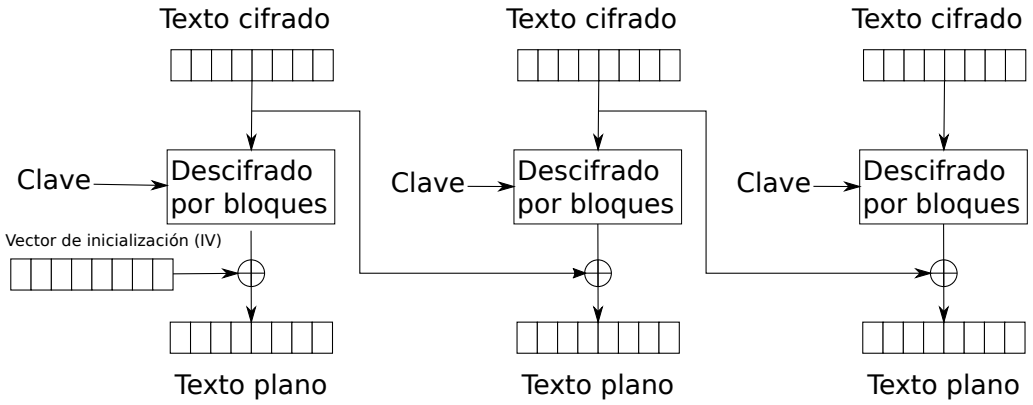

FIGURA 3.6: Descifrado usando Cipher Block Chaining (CBC)

A la hora de desencriptar, la misma función utilizada en la encriptación es usada para obtener cada bloque de texto a partir de los bloques cifrados. Una representación de este proceso lo encontramos en la Figura 3.6.

El objetivo de un **cifrado por bloques** es poder paralelizar el proceso de encriptación. Lamentablemente, el cifrado con **CBC** debe ser secuencial, ya que para obtener cada bloque es necesario haber generado antes el anterior.

Sin embargo, el proceso de desencriptación con **CBC** sí que puede ser paralelizado, ya que las mútiples funciones de descifrado que se realizan no dependen de ningún bloque anterior. *[6]*

3.8. AES

Advanced Encryption Standard (AES), también conocido como **Rijndael** por sus creadores, es el resultado del proceso de búsqueda de un estándar para encriptación por parte del gobierno de los Estados Unidos.

AES es una variante de **Rijndael** (que a su vez es una variante de Square), del cual solo toma algunos modos. Mientras que el algoritmo original puede tomar tamaños

de bloque múltiplos de 32 bits,[5] **AES** únicamente opera con tamaños de bloque igual a 128 bits.

Con el tamaño de las claves ocurre algo parecido, ya que **AES** solo soporta tamaños de 128, 192 y 256 bits, mientras que el algoritmo original soporta unos cuantos más. *[21]*

3.8.1. Estado (State)

Antes de meternos a hablar del algoritmo en sí, es necesario explicar una estructura que se usará constantemente.

El **Estado (State)** es una estructura bidimensional de bytes con la que se operan los bytes de los bloques de entrada. Está compuesto por 4 filas y 4 columnas, dando un total de 32 celdas, en las cuales se almacenan los 16 bytes que forman un bloque. *[13]*

3.8.2. Transformaciones

El algoritmo consta de ciertas fases, una de ellas consiste en someter el *State* a unas cuantas rondas de transformaciones. Estas transformaciones son las siguientes:

- **SubBytes** – Mediante una transformación no lineal, los bytes del *State* son reemplazados por otros usando una tabla de sustitución. (Figura 3.7)
- **ShiftRows** – Los bytes de las 3 últimas filas del *State* se desplazan de manera cíclica, cada fila con un offset distinto. (Figura 3.8)
- **MixColumns** – Mediante una transformación lineal, las columnas del *State* se mezclan para producir unas nuevas. (Figura 3.9)
- **AddRoundKey** – En esta transformación, una *Round Key* se añade al *State* mediante una operación XOR. La itud de la *Round Key* debe ser igual al tamaño del *State*. (Figura 3.10)

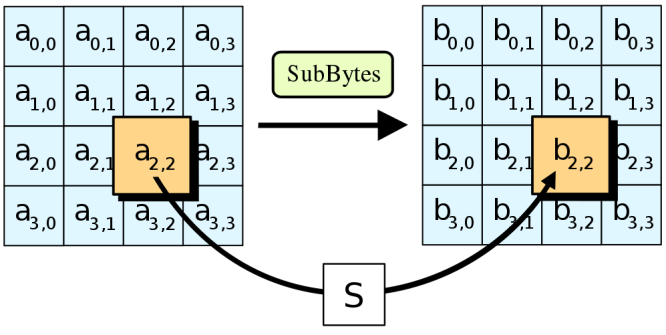

FIGURA 3.7: Operación SubBytes para AES *[18]*

[13]

[5]Con un mínimo de 128 bits y un máximo de 256.

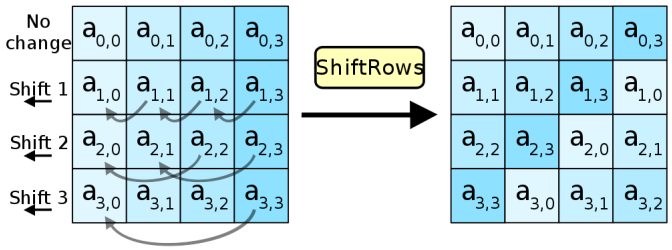

FIGURA 3.8: Operación ShiftRows para AES *[17]*

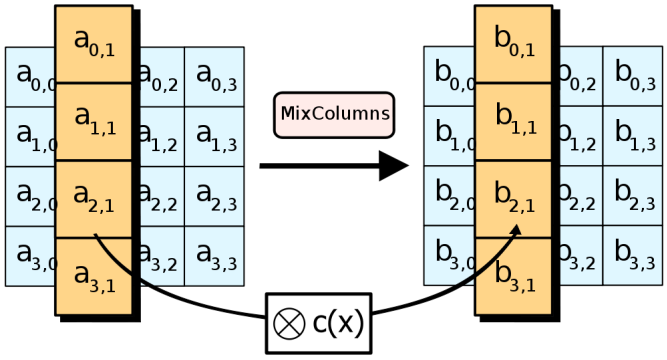

FIGURA 3.9: Operación MixColumns para AES *[16]*

3.8.3. Algoritmo

En **AES** el tamaño del bloque de entrada, del de salida y del *State* es de 128 bits, y el tamaño de la clave puede tomar los valores de 128, 192 ó 256 bits.

Para cada bloque de entrada, la primera etapa del algoritmo consiste en generar las *Round Keys* a partir de la clave, usando el esquema de claves Rijndael.[6]

Una vez conseguidas las *Round Keys*, se pasa por una ronda inicial especial en la cual solo se realiza la transformación *AddRoundKey*.

CUADRO 3.1: Combinaciones para el número de rondas en AES.

Key size (bits)	Block size (bits)	Rounds (Nr)
128	128	10
192	128	12
256	128	14

Después de esta etapa inicial, se pasa a las rondas habladas en el punto anterior. El número de rondas que lleva a cabo el algoritmo depende del tamaño de la clave, lo cual vemos representado en el Cuadro 3.1.

En todas estas rondas menos en la útlima, el orden de las transformaciones será siempre el mismo:

$$SubBytes \rightarrow ShiftRows \rightarrow MixColumns \rightarrow AddRoundKey$$

[6]Este esquema expande una clave en un número determinado de claves separadas.

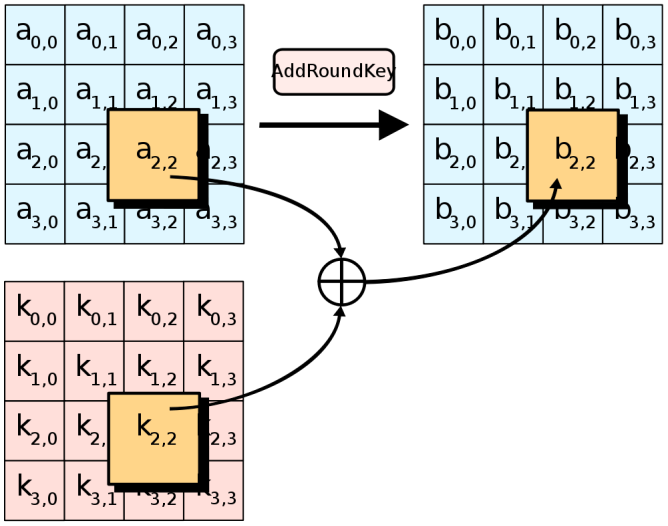

FIGURA 3.10: Operación AddRoundKey para AES *[15]*

La última ronda es idéntica a las anteriores salvo por el hecho de que la transformación *MixColumns* no se realiza.

El resultado de todo esto será el *output* de nuestro bloque de entrada. *[13]*

3.9. Criptografía de clave pública

Uno de los problemas que tiene la **criptografía de clave simétrica** es que usamos la misma clave para el cifrado y el descifrado, por lo que se hace necesaria su ocultación. En la **criptografía de clave pública**, al hacer uso de dos claves, no es necesario mantener en secreto ambas. Aquella que ocultamos es llamada clave privada y la otra, pública.

La esencia de este algoritmo radica en que un mensaje cifrado con una clave pública solo puede ser descifrado con su homóloga privada, y viceversa. De esta manera podemos, tal como su propio nombre indica, difundir públicamente nuestra clave *pública* mientras mantenemos oculta la *privada*.

Si lo que queremos es *confidencialidad* durante la comunicación, entonces encriptaremos el contenido del mensaje con la clave pública del destinatario, de forma que solo él podrá descifrarlo (Figura 3.11).

Por otra parte, también nos interesa que cuando alguien reciba nuestro mensaje pueda estar seguro de que realmente es nuestro. Para lograr esto, lo que hacemos es cifrar un resumen del mensaje (*hash*) con nuestra clave privada. De esta forma, cuando alguien lo descifre con nuestra clave pública y compruebe el resumen, podrá estar seguro de que proviene de nosotros.[7] Así conseguimos preservar la *integridad* y la *autentificación* del mensaje, dos parámetros muy importantes a tener en cuenta en seguridad. *[9]*

[7] Es, en esencia, el fundamento sobre el que se basa la firma digital.

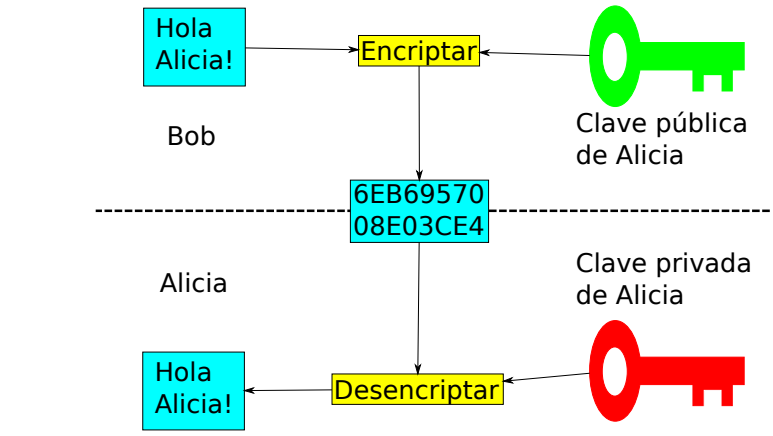

FIGURA 3.11: Esquema general del cifrado de clave pública

3.10. RSA

Uno de los primeros algoritmos de criptografía pública (y el más utilizado hoy) es **RSA** (**Rivest–Shamir–Adleman**)[8]. Una de sus características más distintivas es el problema que plantea factorizar el producto de dos grandes números primos (*factoring problem*). El resultado de esta factorización es luego usado en la generación de las claves.

En general, los algortimos de criptografía de clave pública son bastante más lentos que los de clave simétrica. Por ello, se suelen usar en conjunto con algún algoritmo de criptografía simétrica, como **AES**. *[24]*

3.10.1. Clave pública RSA

Dentro del conjunto del par de claves **RSA**, la *pública* es aquella que no mantenemos en secreto para que cualquiera que quisiera comunicarse con nosotros pudiera hacerlo de manera confidencial. Este tipo de claves consta de dos elementos:

- **n** – el módulo RSA, un entero positivo.
- **e** – el exponente público RSA, otro entero positivo.

Lo primero que se debe hacer para generar la clave *pública* es seleccionar de manera aleatoria dos números primos impares distintos, lo suficientemente grandes como para que sea computacionalmente inviable su descubrimiento por parte de un atacante. Una vez encontrados, el módulo **n** de la clave *pública* será el producto de estos dos números primos.[9]

$$n = p \cdot q$$

Ahora que tenemos el módulo de la clave definido, es hora de generar un exponente público **e** adecuado. Para ello, haremos uso de la función de Carmichael[10], la cual

[8] RSA debe su nombre a sus creadores: Ron Rivest, Adi Shamir y Leonard Adleman.

[9] Por convención, se denotan como p y q.

[10] La función que se muestra solo es válida cuando n es el producto de dos números primos impares y distintos.

tiene la siguiente forma:

$$\lambda(n) = mcm(p-1, q-1)$$

El exponente público **e** será aquel número entero comprendido entre 3 y (**n** - 1) que satisfaga:

$$MCD(e, \lambda(n)) = 1$$

[10]

3.10.2. Clave privada RSA

La clave *privada* es la que mantenemos oculta. Aunque existen varios modelos, generalmente consta de dos elementos:

- **n** – el módulo RSA, un entero positivo (Debe ser el mismo que el de la clave *pública*).
- **d** – el exponente privado RSA, un entero positivo.

Para calcular el exponente privado **d** deberemos elegir un entero positivo menor que **n**, que además cumpla:

$$e \cdot d \equiv 1 \pmod{\lambda(n)}$$

, donde **e** será el correspondiente exponente público. *[11]*

3.10.3. Evaluación de la función RSA

Lo primero que debemos hacer es conseguir una representación de nuestro mensaje como un número entero, comprendido entre 0 y (**n** - 1).

Vamos a suponer que este mensaje queremos enviárselo a alguien cuya clave pública es (e, n). Si nuestro mensaje (recordemos que ahora es un entero) es M, para obtener el mensaje cifrado, aplicaremos la siguiente operación:

$$C \equiv E(M) \equiv M^e \pmod{n}$$

El entero C será del mismo tamaño que M y representará al mensaje cifrado.

Si ahora el destinatario quisiera descifrar el mensaje, solo tendría que revertir la operación con su clave privada (d, n) de la siguiente manera:

$$M \equiv D(C) \equiv C^d \pmod{n}$$

De esta manera, recuperaría el mensaje original.[11] *[14]*

[11] Para cifrar con una clave privada y descifrar con una pública se hacen las mismas operaciones, cambiando e y d donde corresponda.

3.11. RSASSA-PSS

Antes hablamos de la necesidad de preservar la *integridad* y la *autentificación* en las comunicaciones. **RSASSA-PSS** es un algoritmo de firma digital que sirve precisamente para ello. Combina RSA con un método de codificación llamado Probabilistic Signature Scheme (PSS). [12]

3.11.1. Probabilistic Signature Scheme (PSS)

PSS es un método de codificación[13] desarrollado por Mihir Bellare y Phillip Rogaway, los cuales buscaban mejorar los métodos que existían. Para ello, incluyeron en su esquema el uso de una *salt*,[14] lo cual haría más seguro el algoritmo frente a intentos de romperlo. *[2]*

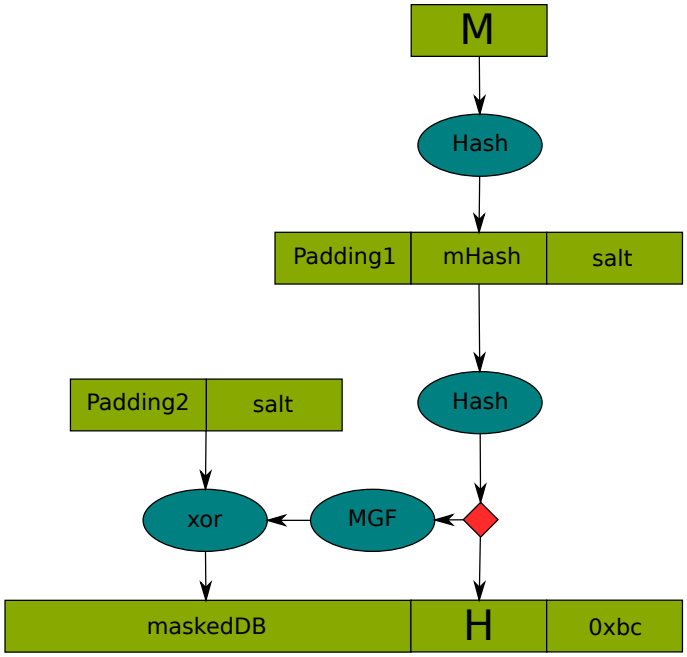

FIGURA 3.12: PSS de acuerdo a PKCS #1 V2.2 / RFC 8017

Como vemos en la Figura 3.12, **PSS** genera un resumen (*hash*) del mensaje. Este resumen se vuelve a pasar de nuevo por una función *hash*[15] junto a un *padding*[16] y la *salt* tal como muestra la figura.

[12] Las siglas SSA corresponden a Signature Scheme with Appendix, lo que quiere decir que la firma va añadida al final del mensaje o junto a él.

[13] Un método de codificación es una operación que permite convertir un carácter de un determinado conjunto en un símbolo de otro sistema de representación.

[14] Bits aleatorios.

[15] Las dos funciones *hash* que se usan en el esquema deben ser la misma.

[16] El primer *padding* estará formado por 8 bytes con valor *0x00*.

El resultado de esta operación será la segunda de 3 piezas que conformarán nuestro mensaje codificado y la denotaremos como **H**. Para la primera (*maskedDB*), generaremos una máscara[17] de **H** y haremos una operación *xor* con ella y la salt (junto a otro *padding*).[18]

Ahora solo nos quedará añadir al final de nuestra salida un byte con valor *0xbc* y ya habremos acabado. *[12]*

3.12. HTTP

Hypertext Transfer Protocol (HTTP) es un protocolo de nivel de aplicación para sistemas de información distribuidos.

Es un protocolo genérico y sin estado que es usado para múltiples tareas como la transferencia de hipertexto o la representación de sistemas de ficheros.

Una característica de HTTP es la negociación de la representación de los datos, lo que permite construir sistemas indpendientemente de los datos que se vayan a transmitir. *[7]*

[17] Una máscara es muy parecida a una *hash*. Mientras la segunda tiene un tamaño determinado, una máscara puede tomar distintos tamaños según la necesidad.

[18] Este *padding* está formado por una cantidad variable de bytes con valor *0x00*, teniendo al final un byte de valor *0x01*.

Capítulo 4

Diseño e implementación

4.1. Arquitectura de seguridad

Para desarrollar este punto, la sección se va a dividir en las distintas propiedades de la seguridad de la información que posee la aplicación:

4.1.1. Confidencialidad

Desde un principio se pensó en criptografía de clave simétrica para el cifrado de la información. Y, debido a su buena reputación, AES como algoritmo de cifrado.

De entre todos los tamaños de clave disponibles se optó por el de 128 bits, ya que ofrecía un nivel de seguridad adecuado para nuestra aplicación. Además, un nivel superior habría supuesto una mayor carga computacional.

Toda este despliegue no tendría sentido si no se incluyese confidencialidad a la clave utilizada. Por ello, se utilizó un sistema de claves asimétricas para proteger la clave simétrica. Debido otra vez a su uso extendido, se utilizó RSA como algoritmo de cifrado.

Ya que el número de operaciones de cifrado asimétrico sería menor que el del simétrico, se decidió utilizar el tamaño máximo para una clave RSA: 4096 bits.

4.1.2. Integridad

También era importante que el contenido del mensaje permaneciese intacto. Para ello había que incluir algún mecanismo de resumen que nos asegurase la integridad del mensaje.

En un principio se barajó utilizar PKCS1 V1.5, ya que me resultaba familiar de la asignatura de seguridad. Sin embargo, algunos profesores me recomendaron utilizar PSS, ya que las prestaciones de seguridad eran mejores.

Efectivamente, el uso de una *salt* en la generación del resumen codificado hacía de PSS un algoritmo mas consistente frente a determinados ataques enfocados en la obtención del mensaje original a partir de su resumen.

4.1.3. Autenticación

El destinatario tenía que estar seguro de que el mensaje venía de la persona que se supone que se lo había enviado. Para lograrlo se incluyó junto al mensaje una MAC.[1]

Se utilizó como algoritmo RSASSA-PSS, ya que combinaba los algoritmos utilizados para preservar la confidencialidad y la integridad. Este algoritmo codifica el mensaje utilizando para ello PSS y luego firma (cifra) el mensaje usando RSA.

Para proporcionarle aún más seguridad a la firma se decidió convertir la MAC en una HMAC, lo que implicó cifrar la MAC con la clave pública del destinatario de forma que solo él pudiera acceder a su contenido.

4.2. Arquitectura del software

Para explicar la arquitectura software de la aplicación, esta sección va a estar dividida en varios prototipos que se han desarrollado:

4.2.1. Shatter 0.1

El primer prototipo de la aplicación se desarrolló enteramente en Java, usando para ello el entorno de desarrollo Eclipse. Esta primera versión buscaba poder dividir un fichero en varios fragmentos de un tamaño dado, y luego poder recomponerlo. Para ello se implementaron algunas clases:

- **Slice** – Un Slice es uno de los fragmentos en los que un fichero original se ha dividido. Está formado por una cabecera (Header) y un array de bytes en el que se almacena el contenido del segmento del fichero. (Figura 4.1)
- **Header** – En esta clase se almacenan los metadatos de los Slices. Se guardan datos como un contador, el número total de Slices para un fichero, un ID para la sesión[2] y el tamaño original del fichero. (Figura 4.1)
- **Slicer** – Esta clase es la *fábrica* de Slices. Recibe un fichero, un tamaño de bloque y un ID para identificar la sesión. Lee del fichero bloques del tamaño indicado hasta alcanzar el EOF y genera un Slice para cada uno de ellos, con una cabecera distinta. (Figura 4.2)
- **Composer** – Si el anterior recibía un fichero y generaba Slices, éste recibe Slices y devuelve un fichero compuesto. Lee uno a uno los Slices que recibe, prestando especial atención a sus cabeceras y si detecta que alguno falta genera un log de errores. (Figura 4.2)

Al tratarse de un prototipo bastante sencillo, no dió muchos problemas, se alcanzaron fácilmente los objetivos buscados.

[1] Una MAC es una porción de información utilizada para autenticar un mensaje.

[2] En esta primera iteración de la aplicación, el ID para identificar la sesión era un resumen Hash del fichero.

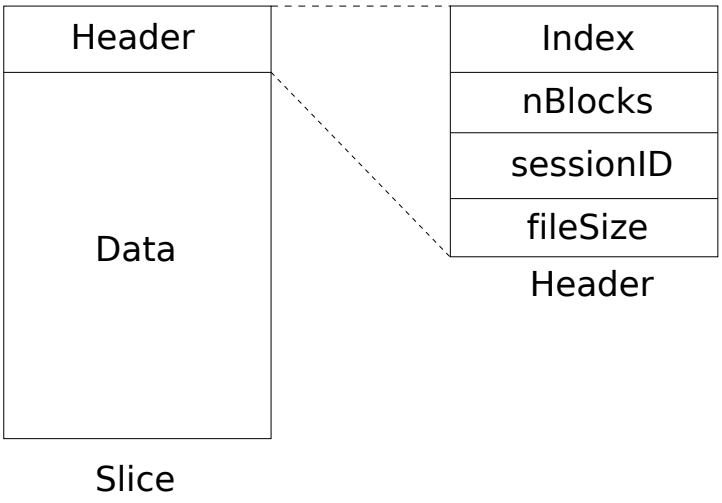

FIGURA 4.1: Esquema general de las clases Slice y Header (Versión 1)

4.2.2. Shatter 0.5

En esta segunda versión, el objetivo era proporcionar confidencialidad a las Slices. Para llevarlo a cabo se implementaron las siguientes clases:

- **EncFile** – Viene a ser una Slice cifrada. A parte de los datos encriptados de la Slice, también incluye una cabecera (EncFileHeader) con algunos datos importantes y una pseudocabecera (FalseHeader) con datos menores. (Figura 4.3)
- **EncFileHeader** – Como decía antes, esta cabecera se usa para almacenar algunos metadatos importantes como, en este caso, el vector de inicialización (IV) que se ha usado para cifrar la Slice. (Figura 4.3)
- **KeyFile** – Esta clase se utiliza para almacenar la clave simétrica que se ha utilizado para crear los EncFiles.
- **AESLibrary** – Básicamente, una clase que se encarga de generar de manera aleatoria y segura claves simétricas y vectores de inicialización.
- **SymmetricCipher** – Esta es la clase que se encarga de hacer la parte más importante en cuanto a la confidencialidad. Una vez que ha sido inicializado con una clave simétrica, genera textos cifrados a partir de texto plano y un IV. Igualmente, puede llevar a cabo el proceso inverso.
- **Encryptor** – La *fábrica* de EncFiles. Genera de manera aleatoria y segura (Utilizando una de las clases mencionadas anteriormente) una clave simétrica para un algoritmo establecido y, con ella, inicializa una instancia de un SymmetricCipher. A continuación se le pasa un array de Slices del cual genera un array de EncFiles, que es el que retorna. (Figura 4.4)
- **Decryptor** – La contraparte del Encryptor. Realiza el proceso inverso y devuelve un array de Slices a partir de uno de EncFiles. (Figura 4.5)

A pesar de que la seguridad que proporciona este prototipo no está completa (Ya que es propenso a algunos ataques), sienta las bases de lo que será la arquitectura de seguridad de la aplicación.

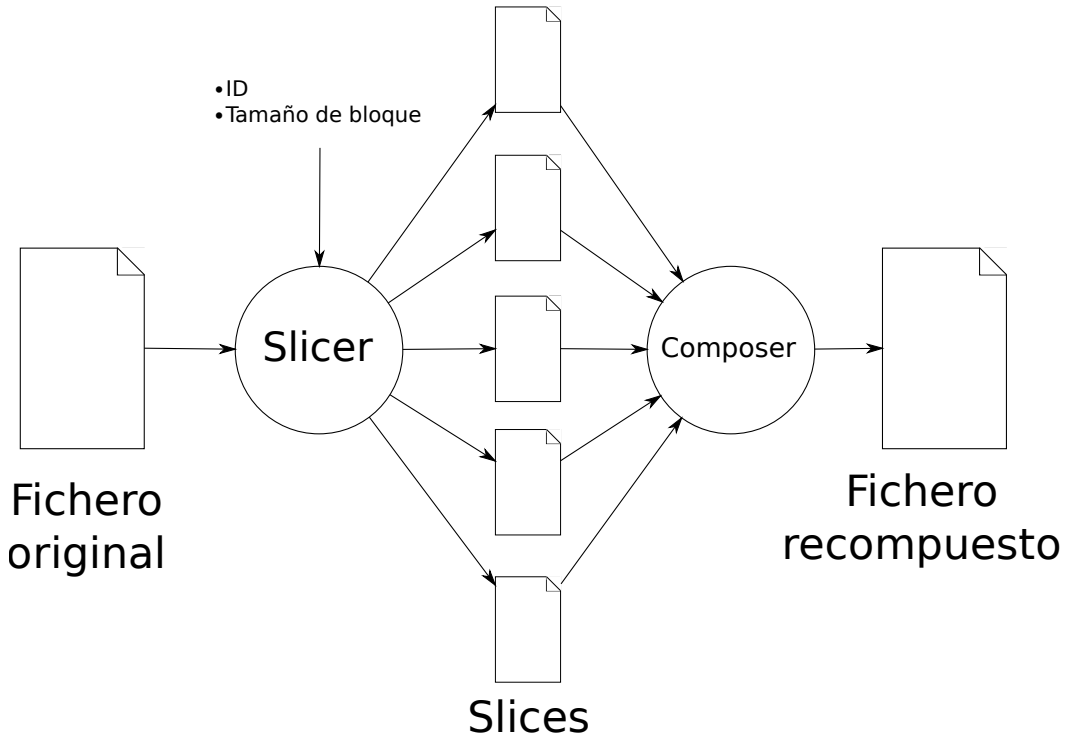

FIGURA 4.2: Esquema general del Slicer y el Composer

4.2.3. Shatter 0.8

A pesar de que ya teníamos confidencialidad en las Slices, había que proporcionársela también al KeyFile. En este prototipo se crearon las siguientes clases para lograrlo:

- **EncKeyFile** – En esta clase se almacena la clave simétrica cifrada usando para ello una clave asimétrica. Tiene una cabecera en la que se guardan algunos datos importantes.
- **EncKeyFileHeader** – La cabecera del EncKeyFile. En ella se almacena una firma cifrada (HMAC) de la clave simétrica sin cifrar.
- **RSALibrary** – Esta clase es la encargada de generar un par de claves asimétricas, de escribirlas y leerlas de un fichero y de cifrar un texto plano a uno cifrado, y viceversa.
- **Signature** – Básicamente una clase para contener una firma.
- **SecureSignature** – Contiene la firma comentada antes pero cifrada usando una clave asimétrica. Es una HMAC.
- **Signer** – La clase que se encarga de recibir Slices, EncFiles, KeyFiles y demás y devolverlos firmados. Asimismo, se encarga de comprobar las firmas de todos estos ficheros para preservar la autenticidad de la información que portan. (Figura 4.6)
- **RSAPSS** – Esta clase incorpora todos los métodos necesarios para generar, a partir de un texto plano, un texto codificado y firmado usando el algoritmo RSASSA-PSS. También realiza el proceso inverso y puede verificar las firmas.

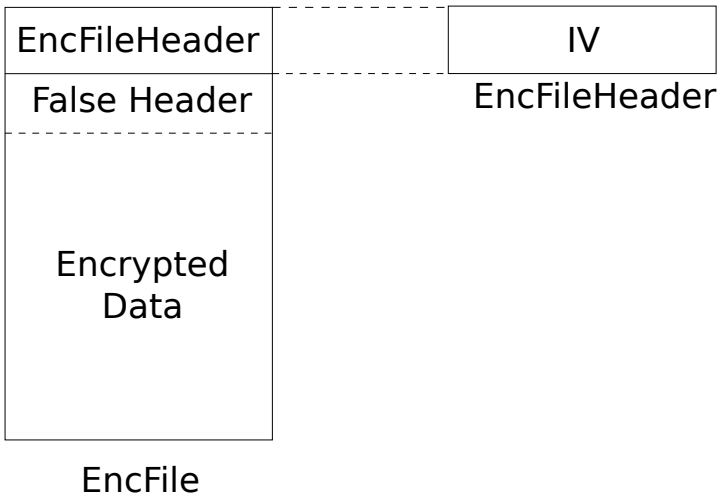

FIGURA 4.3: Esquema general de las clases EncFile y EncFileHeader (Versión 1)

Con el prototipo anterior nos dimos cuenta de que, a parte de que teníamos que generar un modo seguro de transmitir la clave simétrica, también teníamos que proporcionar una manera de comprobar la autenticidad de los datos.

Debido a ello, incluimos una firma en el KeyFile y en las cabeceras de las Slices y EncFiles (Figuras 4.7 y 4.8). Nos dimos cuenta de que algunas firmas iban a ir en claro cuando viajasen de un equipo a otro. Por ello se decidió la creación de una firma cifrada (HMAC), para evitar posibles "mirones".

Además de las clases mencionadas anteriormente, también se desarrollaron otras clases para generar Strings aleatorios, para leer y escribir los distintos ficheros que intervienen en la aplicación, etc.

4.2.4. Shatter 1.0

Esta versión final de la aplicación está marcada por el salto a la plataforma Android. Gran parte del código desarrollado en los anteriores prototipos se siguió utilizando, sin embargo hubo que cambiar algunas clases:

- **KeyStoreHandler** – Esta clase se usa para realizar todas las operaciones necesarias con el KeyStore de Android. Se encarga de almacenar las claves, recuperarlas, usarlas para encriptar, firmar, etc.
- **HTTPClient** – Un cliente HTTP muy sencillo que únicamente realiza peticiones GET.
- **ExternalStorage** – Para interactuar con el External Storage de Android se creó una clase que incorpora algunos métodos para conseguir paths, descriptores de fichero o crear directorios.

A parte del desarrollo de estas clases, se trabajó también en un servidor HTTP dedicado para almacenar los mensajes enviados por los distintos usuarios de la aplicación.

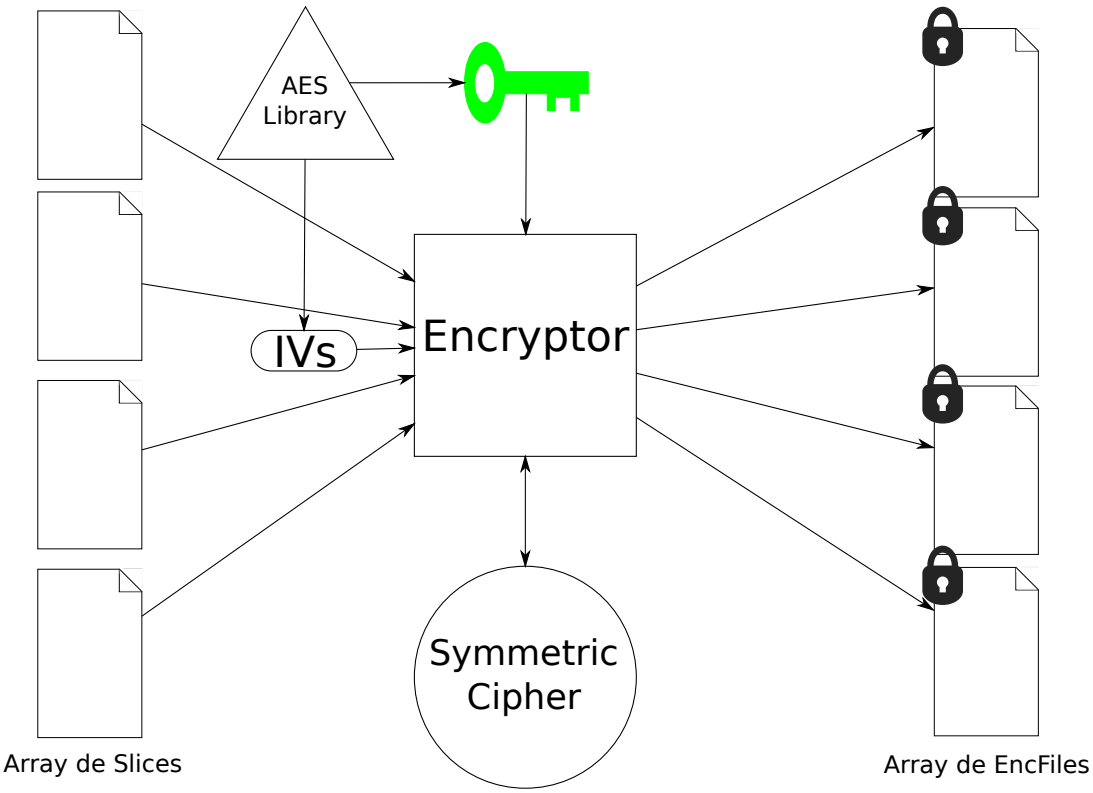

FIGURA 4.4: Esquema general del funcionamiento del Encryptor

Algunas clases, debido al cambio de plataforma, tuvieron que ser eliminadas. El cambio más notorio fue la desaparición de las clases RSALibrary y RSAPSS, ambas sustituidas por KeyStoreManager.

Un cambio menor fue la generación de un ID de sesión aleatorio para las cabeceras de algunas clases (Anteriormente se usaba un resumen hash del fichero).

El esquema general de la aplicación se puede dividir en dos: Una primera parte se encarga de la división y cifrado del fichero (Figura 4.9). La otra se encarga de descargar del servidor los fragmentos cifrados y recomponer el fichero original (Figura 4.10).

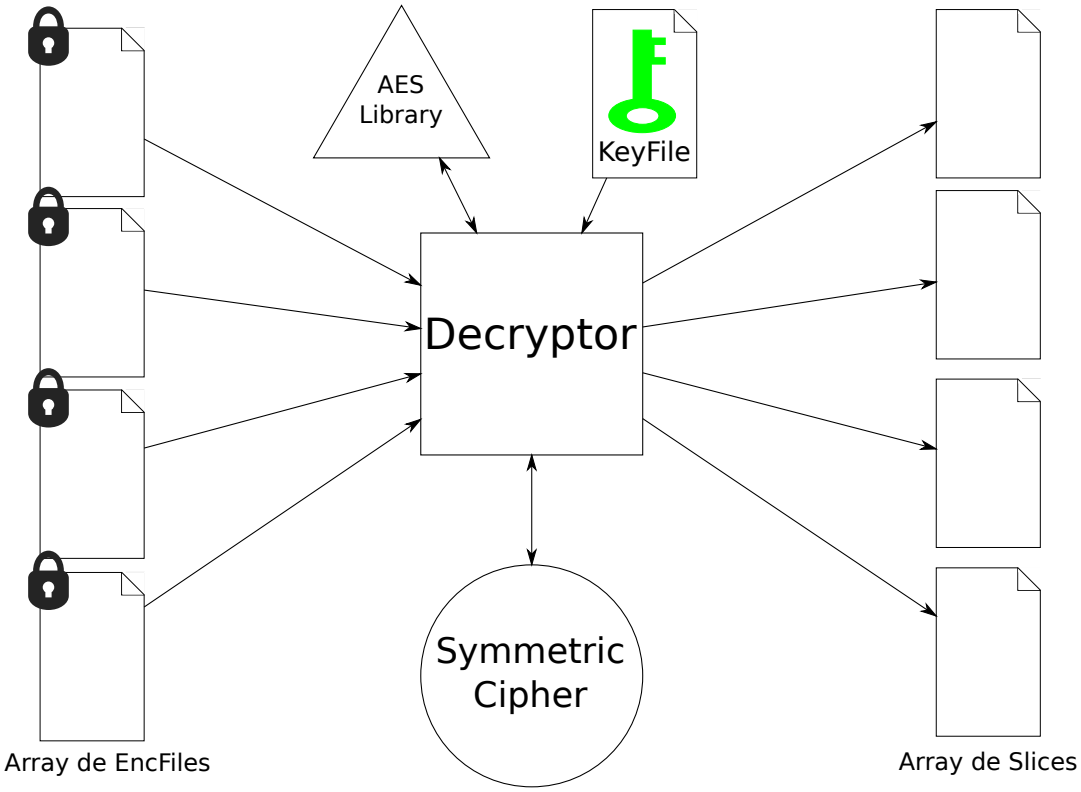

FIGURA 4.5: Esquema general del funcionamiento del Decryptor

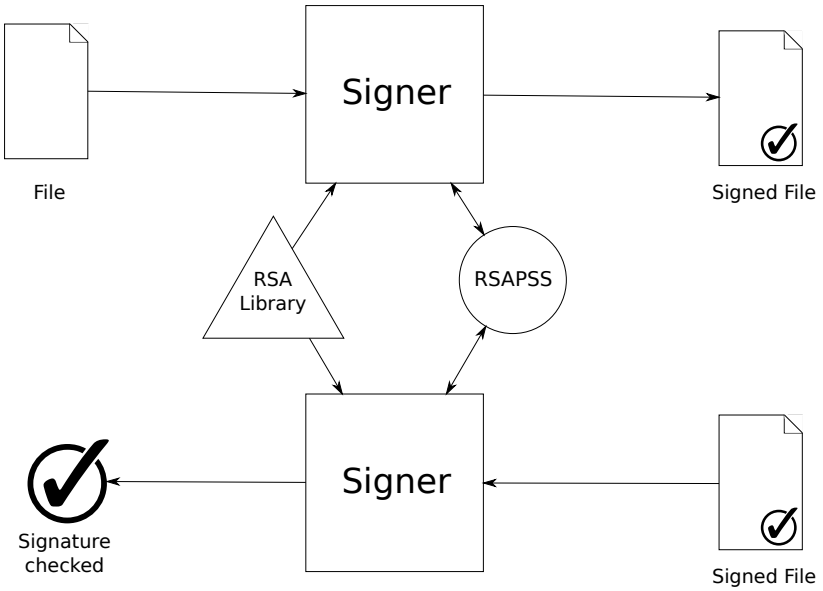

FIGURA 4.6: Esquema general del funcionamiento del Signer

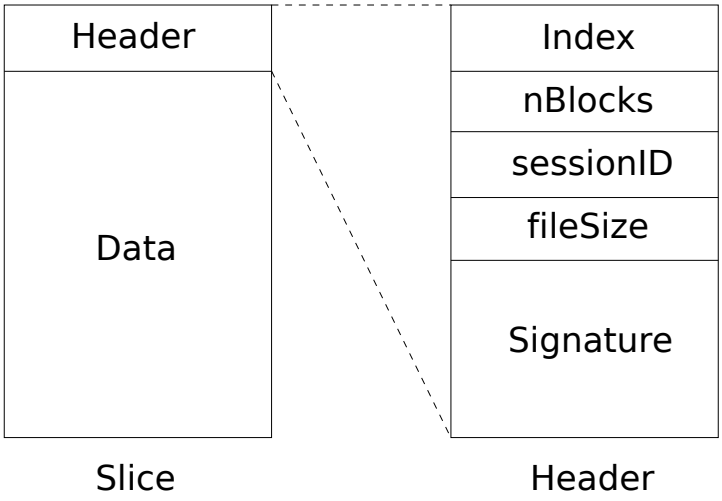

FIGURA 4.7: Esquema general de las clases Slice y Header (Versión final)

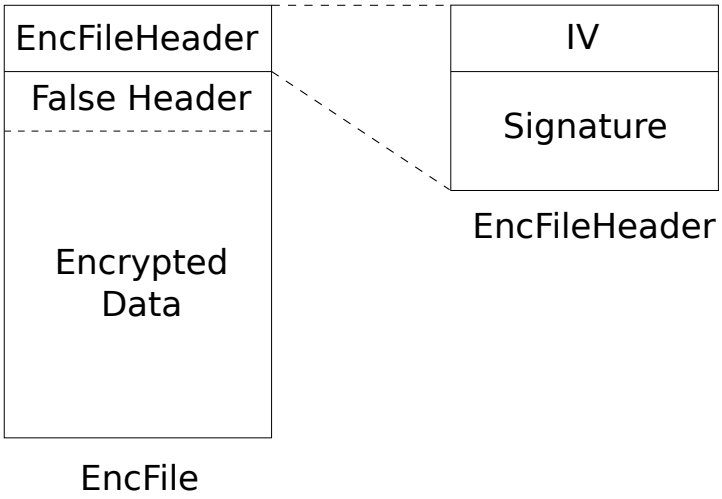

FIGURA 4.8: Esquema general de las clases EncFile y EncFileHeader (Versión final)

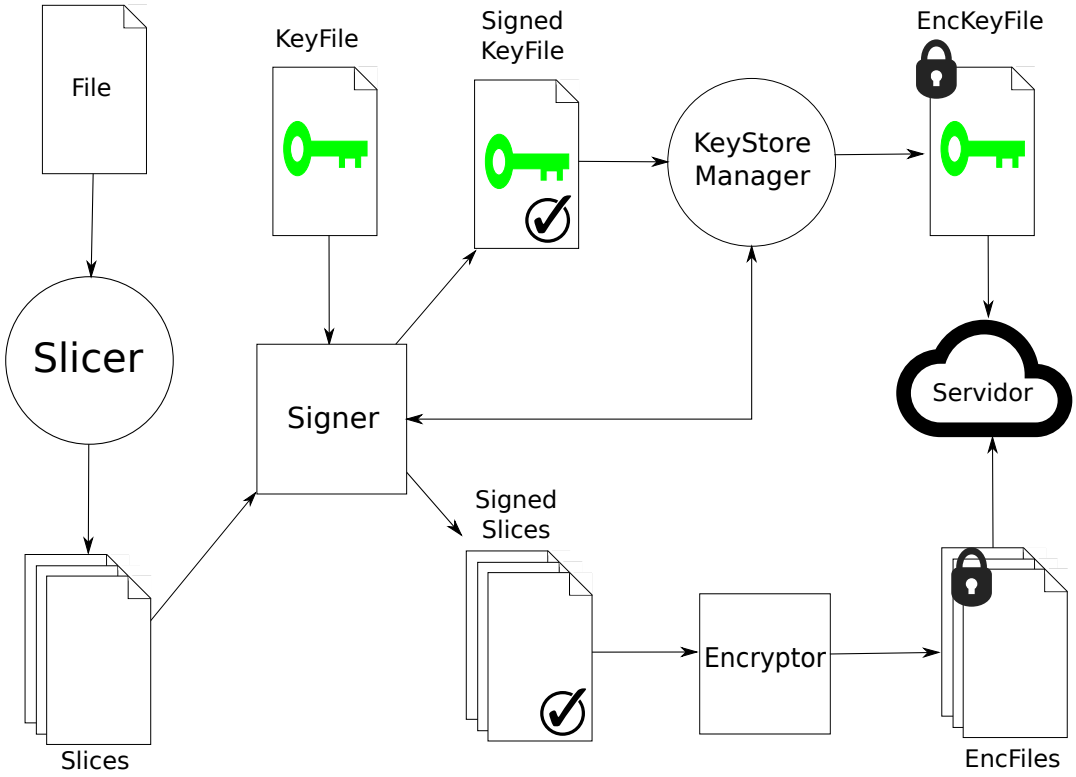

FIGURA 4.9: Esquema general de Slice - Encrypt, la primera parte en el proceso de comunicación de la aplicación

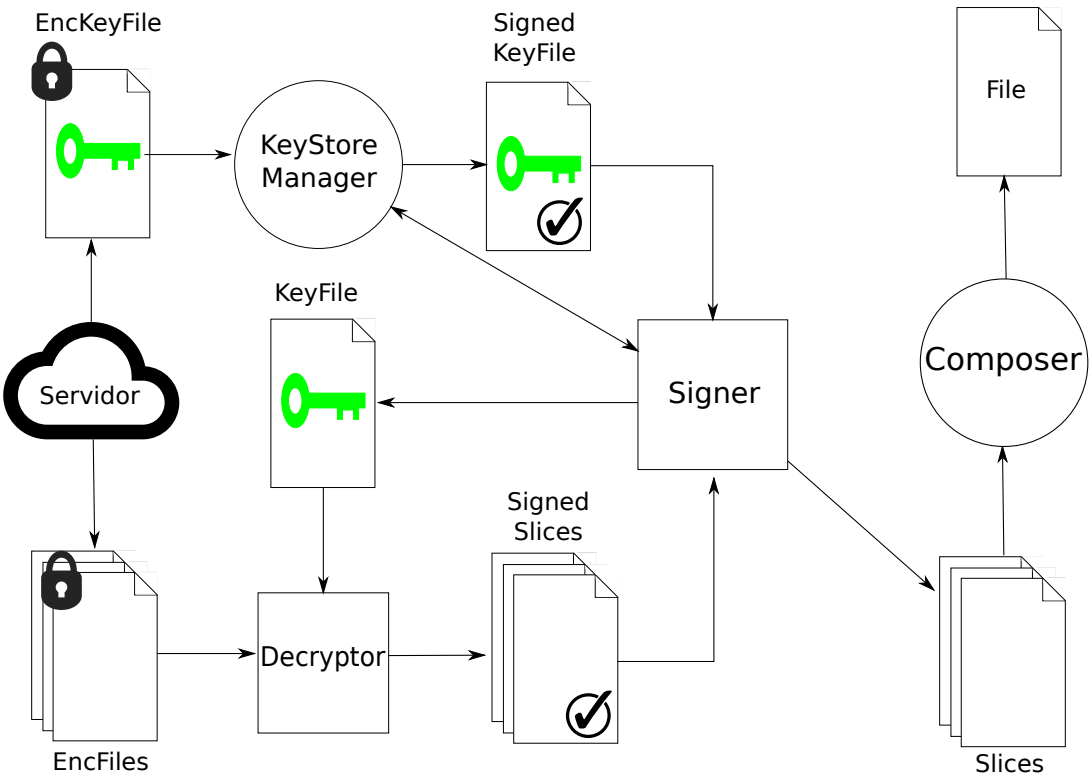

FIGURA 4.10: Esquema general de Decrypt - Compose, la parte final en el proceso de comunicación de la aplicación

Capítulo 5

Resultados

5.1. Ejemplos de utilidad

5.2. Problemas encontrados

Capítulo 6

Conclusiones finales

6.1. Objetivos alcanzados

6.2. Líneas futuras

Bibliografía

[1] Martín Balao. *Criptografía: Padding + ECB + CBC*. 2011. URL: http://martin.com.uy/sec/criptografia-padding-ecb-cbc/.

[2] Mihir Bellare y Phillip Rogaway. *PSS: Provably Secure Encoding Method for Digital Signatures*. 1998. URL: http://grouper.ieee.org/groups/1363/P1363a/contributions/pss-submission.pdf.

[3] The Legion of the Bouncy Castle. *BC Home*. 2013. URL: http://bouncycastle.org/.

[4] Thomas W. Cusick y Pantelimon Stanica. «Cryptographic Boolean functions and applications». En: ed. por Academic Press. 2009. Cap. 7, págs. 158-159.

[5] Hans Delfs y Helmut Knebl. *Introduction to cryptography: principles and applications*. Ed. por Ueli Maurer. Springer, 2007.

[6] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation*. Inf. téc. 800-38A. NIST, 2001. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

[7] R. Fielding y col. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: https://tools.ietf.org/html/rfc2616.

[8] James Gosling y col. *The Java® Language Specification*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[9] Frederick J. Hirsch. *SSL/TLS Strong Encryption: An Introduction*. 2013. URL: http://httpd.apache.org/docs/2.2/ssl/ssl_intro.html#cryptographictech.

[10] Ed. K. Moriarty y col. *PKCS #1: RSA Cryptography Specifications Version 2.2*. Inf. téc. RFC 8017. IETF, 2016, págs. 8-9. URL: https://tools.ietf.org/html/rfc8017#section-3.1.

[11] Ed. K. Moriarty y col. *PKCS #1: RSA Cryptography Specifications Version 2.2*. Inf. téc. RFC 8017. IETF, 2016, págs. 9-10. URL: https://tools.ietf.org/html/rfc8017#section-3.2.

[12] Ed. K. Moriarty y col. *PKCS #1: RSA Cryptography Specifications Version 2.2*. Inf. téc. RFC 8017. IETF, 2016, págs. 42-43. URL: https://tools.ietf.org/html/rfc8017#section-9.1.1.

[13] NIST. *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*. Inf. téc. FIPS 197. NIST, nov. de 2001. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[14] R. L. Rivest, A. Shamir y L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. 1978. URL: http://people.csail.mit.edu/rivest/Rsapaper.pdf.

[15] Wikimedia. *File:AES-AddRoundKey.svg*. 2006. URL: https://commons.wikimedia.org/wiki/File:AES-AddRoundKey.svg.

[16] Wikimedia. *File:AES-MixColumns.svg*. 2006. URL: https://commons.wikimedia.org/wiki/File:AES-MixColumns.svg.

[17] Wikimedia. *File:AES-ShiftRows.svg*. 2006. URL: https://commons.wikimedia.org/wiki/File:AES-ShiftRows.svg.

[18] Wikimedia. *File:AES-SubBytes.svg*. 2006. URL: https://commons.wikimedia.org/wiki/File:AES-SubBytes.svg.

[19] Wikimedia. *File:Android robot 2014.svg*. 2014. URL: https://commons.wikimedia.org/wiki/File:Android_robot_2014.svg.

[20] Wikimedia. *File:Java programming language logo.svg*. 2017. URL: https://en.wikipedia.org/wiki/File:Java_programming_language_logo.svg.

[21] Wikipedia. *Advanced Encryption Standard*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=809193008.

[22] Wikipedia. *Android (operating system)*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Android_(operating_system)&oldid=805639596.

[23] Wikipedia. *Block cipher mode of operation*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=809021817#Common_modes.

[24] Wikipedia. *RSA (cryptosystem)*. 2017. URL: https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)&oldid=806096352.

[25] Wikipedia. *Threat model*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Threat_model&oldid=810785371.